

GPU Accelerated Synchrotron Radiation Calculation

Yong Qin, Ph.D.

High Performance Computing Service (HPCS)
Information Technology Division
Lawrence Berkeley National Laboratory

AFRD Seminar - May 21, 2013

HPCS Overview

- Who are we?



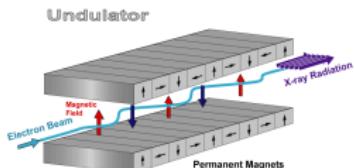
- What do we do?

Introduction

- Spectrum calculation for undulator radiation
- Higher harmonic radiations → more electrons
- More parametric calculations → faster code
 - ▶ Elliptically-polarized undulator (EPU)
 - ▶ Quasi-periodic undulator (QPU)
 - ▶ Micro-bunching effect for coherent radiation
- Far-field model

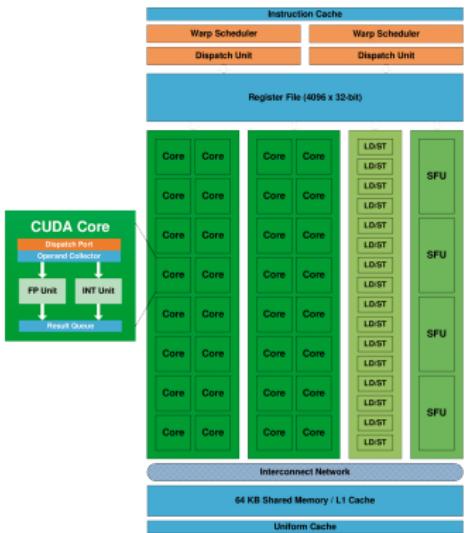
$$\frac{d^2I}{d\omega d\Omega} = \frac{e^2}{4\pi^2c} \left| \int_{-\infty}^{\infty} \frac{\mathbf{n} \times [(\mathbf{n} - \beta) \times \dot{\beta}]}{(1 - \beta \cdot \mathbf{n})^2} e^{i\omega(t - \mathbf{n} \cdot \mathbf{r}(t)/c)} dt \right|^2$$

- Code history: FORTRAN → C# → C++ → C → CUDA C
- Collaborator
 - ▶ Hiroshima Synchrotron Radiation Center (HSRC), Hiroshima University, Japan
 - ▶ Advanced Light Source (ALS), LBNL, USA
 - ▶ High Performance Computing Services (HPCS), LBNL, USA



Target Architecture (Fermi C2050) Overview

- CUDA Capability: 2.0
- 448 CUDA Cores: 14 Streaming Multiprocessors (SM) x 32 CUDA Cores/SM
- 4 SFUs per SM for SP transcendental functions
- L1 Cache (16 KB)/Shared Memory (48 KB) per SM: 64 KB
- 8 Blocks/SM, 1024 Threads/Block, 2 Warp Schedulers/SM, 1536 Threads/SM
 - Fused Multiply-Add (FMA)
 - Peak SP Performance: 1030 GFLOPS
 - Peak DP Performance: 515 GFLOPS
 - Memory Size: 3 GB



Code Analysis

```
// pre-processing
struct RadStep {
    float rx1[NCAL+1], rx2[NCAL+1], ry1[NCAL+1], ry2[NCAL+1];
};
RadStep *RS=(RadStep*)calloc(NUM_PARTICLES,sizeof(RadStep)); // space complexity -> O(N)

// kernel, three nested loops with data dependency
for (int p = 0; p < NUM_PARTICLES; p++) { // NUM_PARTICLES = 1000
    ...
    for (int i = 1; i <= M_field->steps; i++) { // M_field->steps = 2200
        ...
        // 4th-order Runge-Kutta method (data dependency)
        ...
        for (int e = 0; e <= NCAL; e++) { // NCAL = 5000
            ...
            // reduction
            RS[p].rx1[e] = xl * cos(t) * dt + RS[p].rx1[e]; RS[p].rx2[e] = xl * sin(t) * dt + RS[p].rx2[e];
            RS[p].ry1[e] = yl * cos(t) * dt + RS[p].ry1[e]; RS[p].ry2[e] = yl * sin(t) * dt + RS[p].ry2[e];
            ...
        }
    }
}

// post-processing
for (int e = 0; e <= NCAL; e++) {
    for (int p = 0; p < NUM_PARTICLES; p++) {
        ...
        // reduction
        rx1m[e] = rx1m[e] + RS[p].rx1[e]; rx2m[e] = rx2m[e] + RS[p].rx2[e];
        ry1m[e] = ry1m[e] + RS[p].ry1[e]; ry2m[e] = ry2m[e] + RS[p].ry2[e];
        ...
    }
}
```

Solutions

```
// pre-processing
cudaError = cudaMalloc((void**)&(dev_RS_sum[device]), sizeof(struct RadStep)); // space complexity -> O(1)

// kernel, four nested loops with D&C, parallel reduction
#define WRAPSIZE (SHAREDMEM/THREADSPERBLOCK/4/4)
#define WRAPCOUNT ((NCAL+1)/WRAPSIZE)
FLOAT4 RS[WRAPSIZE] = {0}; // thread cache, vector
volatile __shared__ FLOAT4 RS_shared[THREADSPERBLOCK][WRAPSIZE]; // block cache, vector array
int particle = threadIdx.x + blockIdx.x * blockDim.x;

while (particle < nParticle) { // nParticle = NUM_PARTICLES / nDevice
    ...
    for (j = 0; j < WRAPCOUNT; j++) { // WRAPCOUNT = (NCAL + 1) / WRAPSIZE
        ...
        for (i = 1; i <= M_field->steps; i++) { // M_field->steps = 2200
            ...
            // 4th-order Runge-Kutta method (data dependency)
            ...
            for (e = j * WRAPSIZE; (e < (j + 1) * WRAPSIZE) && (e <= NCAL); e++) { // the j-th loop
                ...
                // reduction within a thread using thread cache
                sincos(t, &sint, &cost); // sincos() greatly increases instruction throughput
                RS[n].x += a1 * cost; RS[n].y += a1 * sint; RS[n].z += a2 * cost; RS[n].w += a2 * sint;
                ...
            }
        }
    }
    // parallel reduction across multiple blocks with shared memory
    ...
}

// post-processing (no more reduction)
```

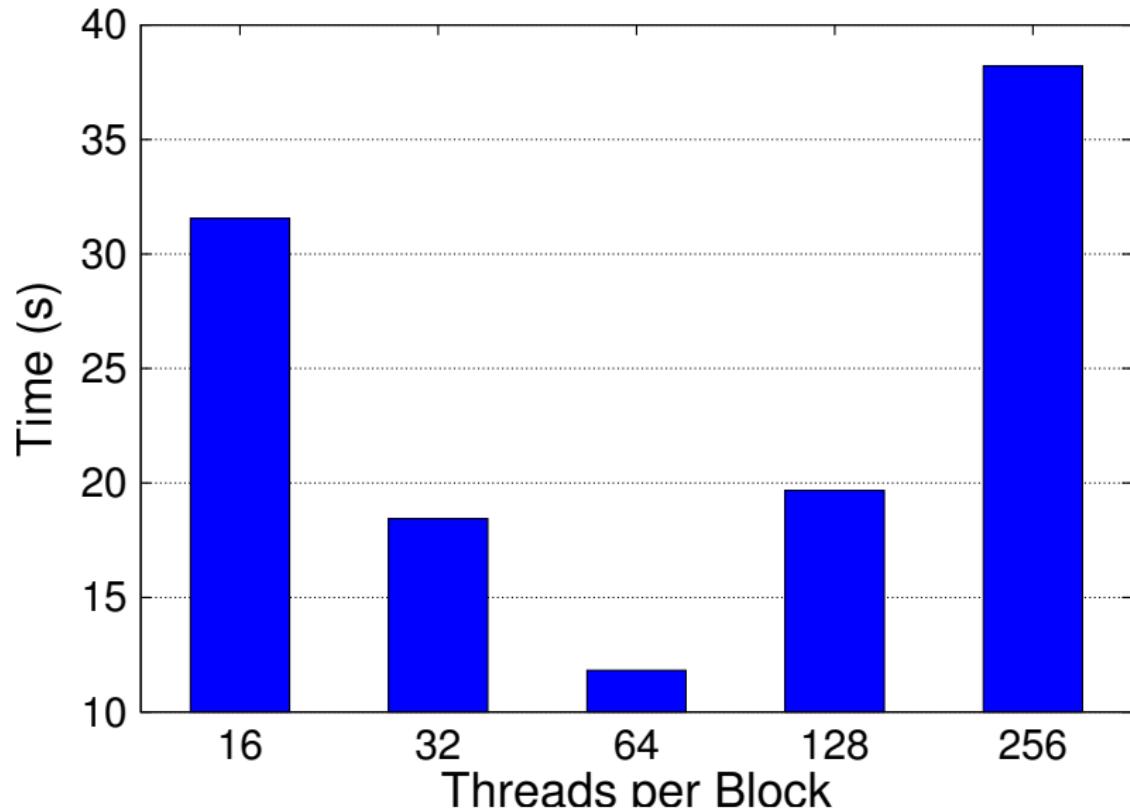


INFORMATION
TECHNOLOGY

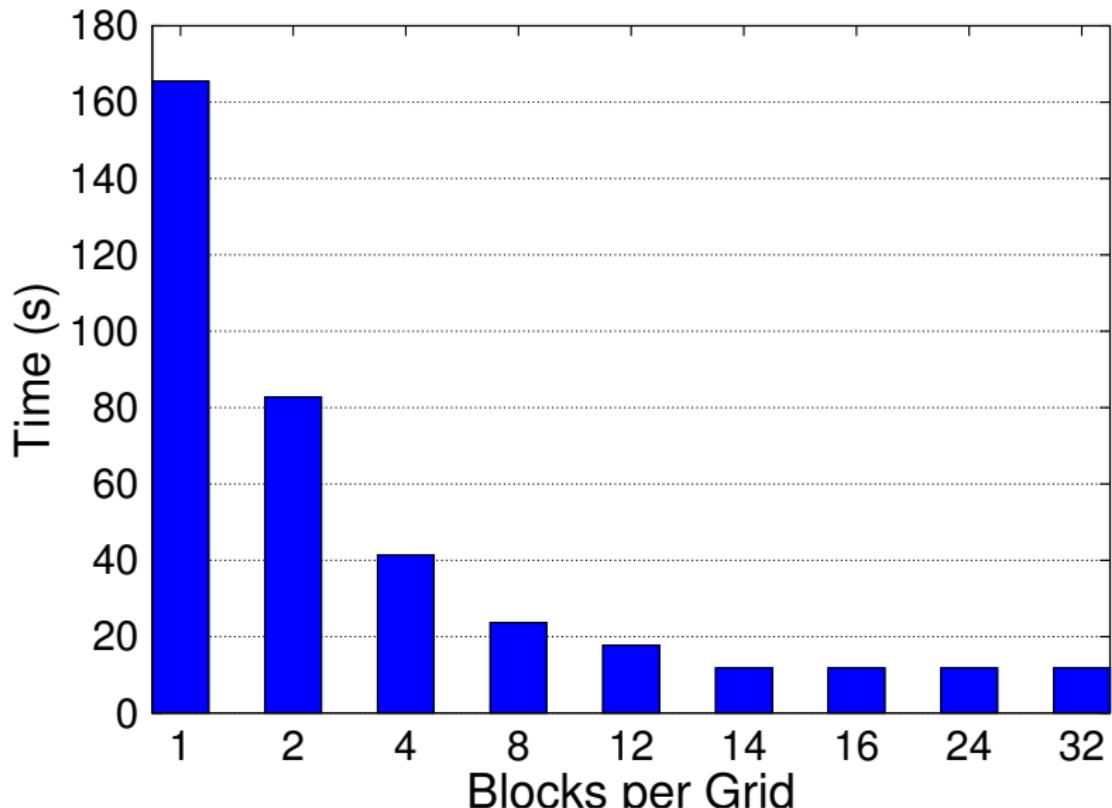
Optimization Strategies

- Instruction throughput analysis
 - ▶ Trigonometric functions in the inner loop limited by SFUs
 - ▶ Combine sin() and cos() into sincos(), $\sim 4x$ throughput increase
- Efficient parallel reduction on GPU
 - ▶ Parallel reduction within a block
 - ★ Divide and Conquer (D&C)
 - ▶ Parallel reduction among blocks
 - ★ Device mutex to prevent race condition
- Memory access optimization
 - ▶ Use local cache improves performance by 37% (reduce bank conflict)
 - ▶ Coalesce memory access saves an extra 10%
 - ▶ Fully utilize shared memory for parallel reduction within a block
 - ★ Reconfigure to 16 KB shared memory and 48 KB L1 cache for D&C
- Optimal Block Size and Grid Size (launch configuration) analysis

Optimal Block Size = 64



Optimal Grid Size = 14



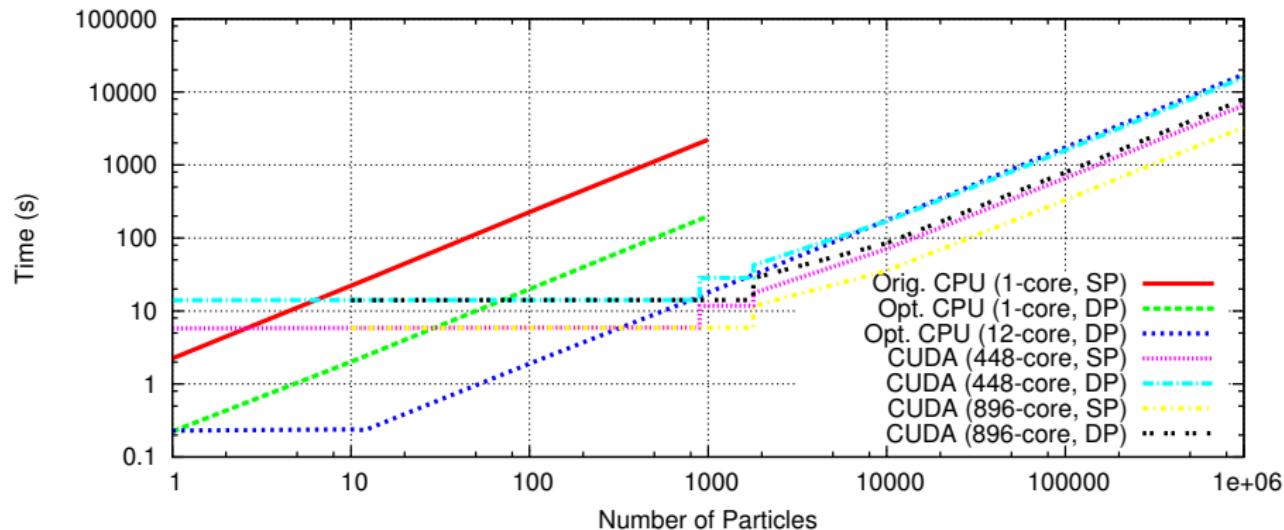
Hardware Configuration & Performance

- Hardware
 - ▶ CPU: 2-socket 6-core Intel Xeon X5650 @ 2.67 GHz
 - ▶ GPU: 2 448-core Nvidia Fermi C2050 @ 1.15 GHz
- Parallel speedups and efficiency

$$S_p = T_1 / T_p, E_p = S_p / p$$

- ▶ SP: $S_{896} = 878$ ($E_{896} = 98\%$)
- ▶ DP: $S_{896} = 890$ ($E_{896} = 99\%$)
- Occupancy = 4.2%
- L1 cache hit ~100%

Performance Comparison



- Opt. CPU (1-core, DP) vs Orig. CPU (1-core, SP): 11x
- Opt. CPU (12-core, DP) vs Orig. CPU (1-core, SP): 123x
- CUDA (448-core, SP) vs Orig. CPU (1-core, SP): 187x
- CUDA (448-core, DP) vs Orig. CPU (1-core, SP): 78x
- CUDA (448-core, SP) vs Opt. CPU (12-core, DP): 2.6x
- CUDA (448-core, DP) vs Opt. CPU (12-core, DP): 1.1x
- CUDA (896-core, SP) vs Orig. CPU (1-core, SP): 374x
- CUDA (896-core, DP) vs Orig. CPU (1-core, SP): 156x

Questions?

